Line Clustering and Contour Extraction in the Context of 2D Building Plans

Andreas Pointner[1], Christoph Praschl[1], Oliver Krauss[1],
Andreas Schuler[1,2], Emmanuel Helm[1,2] and Gerald Zwettler[1,3]
firstname.lastname@fh-hagenberg.at

[1]Research Group Advanced Information Systems and Technology, Research and Development Department, University of Applied Sciences Upper Austria
Softwarepark 11
4232, Hagenberg im Mühlkreis, Austria

[2]Department of Medical and Bioinformatics, School of Informatics, Communications and Media, University of Applied Sciences Upper Austria
Softwarepark 11
4232, Hagenberg im Mühlkreis, Austria

[3]Department of Software Engineering, School of Informatics, Communications and Media, University of Applied Sciences Upper Austria
Softwarepark 11
4232, Hagenberg im Mühlkreis, Austria

ABSTRACT

For the purpose of analyzing a building according to its accessibility or structural resilience, printed 2D floor plans are not sufficient because of the missing link to semantic information. This paper tackles this issue and introduces a concept for clustering classified lines of a floor plan and for creating semantically enriched contour elements based on different image processing, computer vision and machine learning algorithms. Based on a general line clustering approach, we introduce type specific methods for *walls*, *windows*, *doors* and *stairs*. The resulting clusters are in turn used for a contour creation, which uses minimal rotated rectangles. Those rectangles are transformed to polygons that are refined using post processing steps. The approach is evaluated via positive testing using a pixel-based comparison of the process's result. For this, automatically generated as well as real world building plans are used. The final evaluation shows, that the concept reaches a confidence of >90% for door, stair and windows and only around 10% for stairs with the run-time linearly scaling with the size of the input.

Keywords

Clustering, Contour Extraction, Building Plan, Image Processing, Machine Learning

1 INTRODUCTION

In architecture, floor plans are a well-established concept to get a basic understanding about a building structure. This allows to quickly get an overview of size ratios, rooms and the general layout of a building. Despite this, there are a few scenarios where a 2D floor plan is not sufficient. Consequently, it is necessary to transform the plan into a 3D representation to be able to e.g. place furniture in a room or to take a virtual tour through the entire building using a head-mounted display. This paper does not address such digital, semantically rich floor plans, but printed versions, which no longer include that much meta-information.

We have developed a transformation process that uses an image as input, extracts all lines contained in it and classifies those into the types *door*, *wall*, *window* or *stair* using a neural network. These classified lines get clustered as contour elements, that are in turn used to create a 3D model. The final model is used for additional analysis e.g escape routes, or the building's accessibility. This paper focuses on one part in this pipeline; the transformation process that consists of the classified line clustering and the contour creation. This is targeted by our research question: How can classified lines be transformed to contour elements using clustering approaches? The research question leads to the paper's contributions:

- Introducing a concept for clustering classified building plan lines and
- Creating contour elements of the clusters represented as polygons

The remainder of this work outlines an overview of related work on automatic transformation of 2D building plans into 3D models in section 2. In section 3 the background of this work in terms of the used algorithms is presented. Section 4 describes our novel approach of combining different algorithms from the field of image processing, computer vision and machine learning for the transformation of a set of classified lines to contour elements. This transformation process is evaluated in section 5 based on a case study, that utilizes an automatic building plan generation process for creating test data. The results of our experiment are presented in section 6 and are finally concluded in section 7 with some final remarks and possible future work.

2 RELATED WORK

Liu et al. [Liu2017] present a concept on how to transform a rasterized floor plan into a vector-graphics representation which can further be converted to a popup 3D building plan. Consequently, they use predefined shape models, to detect different corner points. In contrast to their work, we do not have any default plan structure, and provide a generalized concept, that works on every building plan within the required visual appearance of the elements.

Zhiliang et al. [Zeng2019] show a deep-learning based system to recognize floor plans. They use a multitask neuronal network with two stages. In the first one, they predict room-boundary elements like doors, windows and walls and in the second task they predict the type of the room. Their work differs from our work w.r.t. the input data, utilizing classified lines and furthermore focusing on the clustering of those. In contrast, their approach combines both steps into one, but on building plans with simple wall types, where each wall is represented by a single line, instead of blocks with multiple filling patterns inside.

Lewis and Séquin [Lewis1998] describe a system for automatically converting 2D floor plans into 3D representations. The presented Building Model Generator (BMG) system takes common CAD formats and transforms them into an internal model, which can be used for adaptions of the plan and analysis. This system is required since the 3D generation of professional CAD tools does not fix faults in the plan. For this reason, the transformation process of BMG also applies corrections to the internal model in terms of geometrical inconsistencies, e.g. not perfectly matching wall corners to improve the results of analysis tasks. The derived model is extruded to a user defined height and elements as doors and windows are adapted to a default height and z-position. The approach of Lewis and Séquin differs to our work in the way that we don't rely on digital CAD data but offer a possibility to create a 3D model from scanned plans.

Stojanovic et al. [Stojanovic2019] are using 3D point clouds to generated 2D floor plans as well as 3D meshes. Their approach is similar to our in the way, that they use clustering techniques to find the relevant contours, but differs by the fact, that they are only focusing on wall elements, whereas our approach deals with windows, doors and stairs as well.

Gerstweiler et al. [Gerstweiler2018] present an approach to create 3D building plans out of 2D floor plans. In contrast to our work they rely on models and heuristics to extract the structural elements, whereas our work uses clustering and contour extraction methods.

Yin et al. [Yin2009] compare different approaches on how to transform floor plans into 3D models. They do not only compare fully automated concepts, but also take a look at semi-automated processes. So et al. [So1998] present a semi-automated way for reconstructing 3D virtual buildings from 2D architecture floor plans. In contrast to our work, they focus on the general concepts and provide mathematical concepts on how to extract poly-lines from walls. Lu et al. [Lu2007] focus on the automatic analysis and integration of architectural drawings. They present concepts for recognizing typical structural objects and architectural symbols and reconstruct the original 3D model out of it. In contrast to our concept, they use predefined shapes for extracting wall lines followed by extracting other elements based on the detected wall lines. Further studies in this field are done by: Dosch et al. [Dosch2000], Ah-Soon and Tombre [AhSoon2001] and Or et al. [Or2008].

3 BACKGROUND

To achieve clustering of linear structures, several algorithmic concepts have been available in the image processing, machine learning and computer graphics domain since decades that are utilized and re-combined in this paper.

While the detection of linear structures can be achieved by edge detection, e.g. applying a Sobel high-pass filter, followed by line analysis in Hough space [Duda1972], perfectly applicable to detecting an ID card in images for scale determination [Pointner2018]. Nevertheless, thereby the continuous line definition has to be locally restricted which is then generally hard to achieve in 2D building construction plans with generally short lines at small construction elements. To detect or prolongate smaller line segments, local hysteresis concepts known from Canny edge detection [Canny1986] or the Bresenham line algorithm [Bresenham1965] known from rasterization in the computer graphics domain is applicable. Utilizing the Bresenham algorithm, a set of 2D pixels is transformed into a continuous line segment at linear run-time complexity without need for time-consuming PCA (principal component analysis)

and SVP (singular value decomposition) [Lee2006]. In the work of Von Gioi et al. [VonGioi2012] a sophisticated line detector is introduced, analyzing the local gradients and level-line fields for sub-pixel accuracy detection of continuous lines on a discrete pixel raster.

To chain up smaller line segments in an iterative manner, the orientation, location and extent of the particular line can be derived by the Minimal Rotated Rectangle algorithm [Eberly2015]. Initial bounding areas are thereby implicitly calculated utilizing the rotating calipers [Preparata1985] algorithm.

For structural analysis of lines in terms of graph analysis, only the mid course at thickness of 1px is required. While for input image gradients detected by Sobel filtering this is achievable by Canny line detection Canny edge detection [Canny1986], for existing binarized line segments generally showing a $width > 1px$, a common thinning approach is applicable, e.g. Zhang-Suen thinning [Zhang1984] constructing the line course from the derived skeleton then or by utilizing stochastic randomized erosion as thinning [Zwettler2010] for speed-up on large images and volumes.

For assembling pre-processed geometric structures such as lines the common vector-based classification algorithm k-Means clustering is applicable, too. Thereby, the clusters are refined in several iterations, where the clusters' centroids are updated and utilized for re-associating the data tuples. This process is repeated until the result converges and there is no re-association of any data tuples anymore or the maximal number of iterations is reached. Due to the randomly selected seed points it is not ensured that the algorithm finds the best solution [Macqueen1967]. To overcome this known limitation of k-Means clustering, the k-means++ algorithm [Arthur2006k] is applicable.

4 APPROACH

We present our clustering approach together with methods to transform *door*, *wall*, *window* and *stair* lines. Additionally, we present the case study, which is used for the actual evaluation of the transformation process. We are using a comma separated values (CSV) representation as input of our approach. The given data is defined for every line by five columns including the x- and y-coordinates of the startpoint, as well as the coordinates of the endpoint. In addition to that, the type of the line is also specified as *door*, *stair*, *wall*, *window* or *unknown*. The approach results in a set of typed contours that are represented by polygons.

4.1 Transformation Approach

The transformation process starts with general input steps that are used for a type-based pre-clustering of all given lines. These general steps consist of reading the

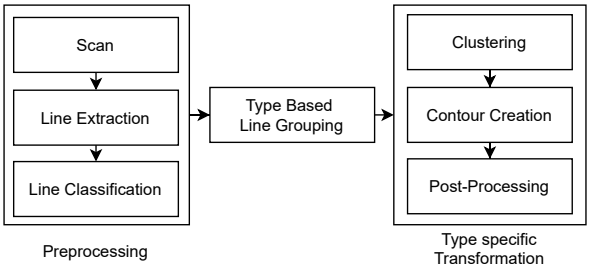


Figure 1: The base process with the preprocessing on the left side and the concept of a type specific processing on the right one.

CSV input, removing all lines typed as unknown and collecting all type equal lines. After that, (1) the actual type specific clustering, (2) the conversion into a 2D contour representation and (3) post-processing steps for removing overlapping wall and window lines are performed. This process is shown in figure 1. For those three steps, there are also some default approaches that are used on the type-pure line collections (created with algorithm 1), before the type specific steps are executed.

Algorithm 1: `getInput()`: Reads lines from CSV (csvFile) and clustering the lines by its type based on a given distance ratio.

```
Data: csvFile, ratio
Result: typed pure line collections
1 minDistance ← sizeOfPlan(lines) * ratio
2 lines ← readCsv(csvFile)
3 linesPerType ← map(type, lineList)
4 forall line in lines do
5     if line.type ≠ Unknown then
6         add(linesPerType[line.type], line)
7     end
8 end
```

For step (1), the line clustering, the method initially performs the Bresenham algorithm [Bresenham1965]. The resulting points are used to determine the distance between two lines of the given collection. If any two points of different lines are within a given distance threshold, the lines are grouped into a cluster. This process is shown in algorithm 2 and is repeated recursively until all input lines are associated to a line cluster.

Step (2), the default conversion method between a line cluster and a contour, is achieved by creating a minimal rotated rectangle of the clusters' point cloud, that consists of all start- and endpoints of the grouped lines. This is shown in algorithm 3.

Finally, the post-processing steps (3), shown in algorithm 4, for realigning overlapping windows or walls are applied.

Algorithm 2: `defaultClustering()`: Clustering of lines by the distance between the two closest points of two lines using a list of all lines, a list of already used lines, the current line and the cluster, as well as the building plan size.

```
Data: allLines, usedLines, currLine, cluster, size
Result: current processed cluster is initialized
if contains(usedLines, currLine) then
    return
end
add(usedLines, currLine)
add(cluster, currLine)
threshold ← sqrt(size.width * size.height) / 30
currPoints ← bresenham(currLine)
forall line in allLines do
    if notContained(usedLines, line) then
        linePoints ← bresenham(line)
        if smallestDistance(currPoints, linePoints)
          < threshold then
            defaultClustering(allLines, usedLines,
              line, cluster, size)
        end
    end
end
```

Algorithm 3: `defaultContourCreation()`: Creating contours of elements by using a Minimal Rotated Rectangle around the lines of the given clusters.

```
Data: clusters
Result: creates map of polygonal contour elements
contourElements ← list()
forall cluster in clusters do
    points ← list()
    forall line in cluster do
        add(points, line.start)
        add(points, line.end)
    end
    contour ← minimalRotatedRectangle(points)
    contour.type ← cluster.type
    add(contourElements, contour)
end
```

The method starts by checking if a *window*/*wall* contour is intersecting with any other *window*/*wall* contour. When the condition applies, the algorithm checks if the orientation of both contours is similar, too. As all contours are defined by rotated rectangles the orientation of them is available, too. This second check is used to avoid information loss by manipulating L- or T-formed elements as e.g. corner windows. For similarly orientated typed contours, the algorithm adapts both elements to the same height and aligns those on the virtual center line defined by the elements' center points.

Algorithm 4: `defaultPostProcessing()`: Window/Wall contour post-processing that aligns two neighboring elements by adjusting their height and rotation based on two lists of contours, one for the windows and the other for the walls.

```
Data: windows, walls
Result: realigned window/wall contours
contours ← combine(windows, walls)
forall contour1 in contours do
    forall contour2 in contours do
        if contour1.intersects(contour2) and
          areSimilar(orientation(contour1),
          orientation(contour2)) then
            height ← average(contour2, contour1)
            scaleTo(contour2, height)
            scaleTo(contour1, height)
            centerline ← line(center(contour2),
              center(contour1))
            orientation ← orientation(line)
            rotateTo(contour2, orientation)
            rotateTo(contour1, orientation)
        end
    end
end
```

In addition to the realignment of the contours, there is also a post-processing step for separating overlapping window and wall elements, that is shown in algorithm 5. This process checks every window for containment in a wall. In that case, the shorter sides of the window are extended until they are intersecting with two sides of the enclosing wall element. This process is shown in figure 2. The intersection points are used to split the wall elements into three parts, where part *A* is defined by the original top-left rectangle corner, the top-left intersection point, the bottom-left intersection point and the bottom-left original rectangle corner. The second part *B* is defined by the four intersection points and the third part *C* consists of the top-right intersection point, the top-right rectangle corner, the bottom-right rectangle corner and the bottom-right intersection point. The two outer parts *A* and *C* are used as new wall elements and the intersection area *B* is removed from the result.

4.1.1 Door specific approach

Since doors are represented as arcs in building plans, it is necessary to convert this representation to a rectangular form. In addition, doors are always surrounded by walls or windows. With this precondition, the transformation approach for door lines is based on the default clustering process and reuses its result to fit doors between surrounding wall or window clusters.

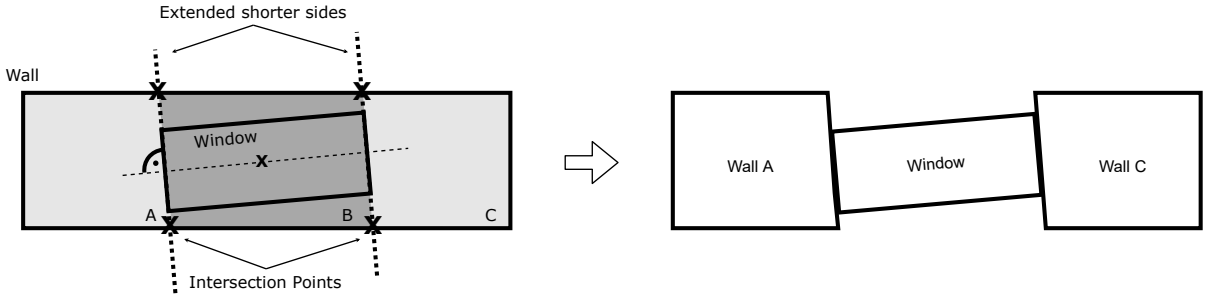


Figure 2: Process of separating a window's contour, that is contained in a wall's contour, by extending the window's shorter sides and using the intersections for cutting the wall element into three parts *A* (light-grey left side), *B* (dark-grey middle) and *C* (light-grey right side), where *B* is removed from the result.

Algorithm 5: `windowWallSeparation()`: Separation of overlapping window and wall elements, by removing the intersection area, based on two lists of contours, one for the windows and the other for the walls.

```
Data: windows, walls
Result: final wall contours
finalWalls ← list()
forall wall in walls do
    forall window in windows do
        if isContainedIn(window, wall) then
            sides ← window.shorterSides
            intersectionPoints ← intersection(sides,
              wall)
            while size(intersectionpoints) < 4 do
                extend(sides)
                intersectionPoints ←
                  intersection(sides, wall)
            end
            toRemove ←
              polygon(intersectionPoints)
            wallParts ← cut(wall, toRemove)
            add(finalWalls, wallParts[0])
            add(finalWalls, wallParts[2])
        else
            add(finalWalls, wall);
        end
    end
end
```

Algorithm 6: `doorClustering()`: Door specific clustering approach, that uses the two closest non door clusters (walls, windows) to find anchor points for the position of the pre-clustered door clusters using the default approach.

```
Data: doors, walls, windows
Result: door clusters
result ← list()
wclusters ← combine(walls, windows)
forall door in doors do
    nearestClusters ←
      getTwoNearestClusters(door, wclusters)
    nearestLine1 ← getNearestLine(door,
      nearestCluster[0])
    nearestLine2 ← getNearestLine(door,
      nearestCluster[1])
    doorCluster ← minimalRotatedRectan-
      gle(nearestLine1.center, nearestLine2.center)

    notOverlapping ← true
    forall cluster in wclusters do
        if overlapping(doorCluster, cluster) then
            notOverlapping ← false;
        end
    end
    if notOverlapping then
        add(result, doorCluster)
    end
end
```

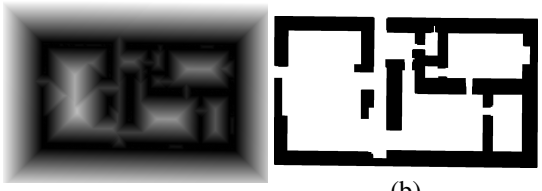
This is done by finding the two nearest lines from two different clusters. The lines' mid points are used as anchor points for the recreated door elements, which is done by fitting in a minimal rotated rectangle between those points. Since the algorithm may take incorrect lines for defining the anchor points a check verifies if the created door is overlapping any other contour. In such a case, the door is considered invalid and is removed from the result. The door specific approach is shown in algorithm 6.

4.1.2 Stair specific approach

Due to the fact that all lines of a stair are somehow connected but different stairs are separated by distance, the characteristics of stairs on building plans are quite unique. For this reason the default clustering and contour conversion approach are sufficient and it was not required to implement a stair specific approach.

4.1.3 Wall specific approach

The default clustering approach is not working for walls because nearly all wall lines are connected and it would result in a few, large clusters. Consequently, the default clustering algorithm is not applicable. Instead, the wall clustering (shown in algorithm 7) uses different image processing approaches. This process starts by creating a distance map (shown in figure 3 (a)) using all wall lines. On the resulting distance image a threshold filtering is applied (shown in figure 3 (b)), with a threshold factor based on the building plan's size. The thresholded result is further manipulated using the Zhang-Suen thinning algorithm [Zhang1984] (shown in figure 4). On the resulting skeleton image, a line segment detector is applied. These skeleton lines are then used as seed points for the clustering process. We create bounding boxes around the extracted skeleton lines and all contained input lines are added to that cluster. The size of the created bounding boxes is in turn normalized based on the size of the building plan. This process is repeated until all input lines are associated with a cluster. Finally, all clusters are checked for intersection. In the case that a cluster is completely contained in another one, these two clusters are merged.



(a) (b)

Figure 3: (a) The calculated distance map and (b) the distance map with an applied threshold.

In addition to the adapted clustering process, the wall specific approach also contains additional contour post-processing steps. Those steps are used to remove clusters which an area smaller than a given threshold that is determined by the overall size of the building plan, and merging cluster that are overlapping with at least 40%. The first of the two steps uses the area of the rotated rectangles and checks if it is greater than a threshold, which is a ratio based on the total building plan size. If the area is below the specified threshold, the contour is removed from the result. For the merging step, the size of the intersection of two overlapping clusters is utilized to determine whether the clusters are merged or not. If the intersection concerns $> 40\%$ of one the clusters, the two clusters are merged. This wall post-processing is shown in algorithm 8.

4.1.4 Window specific approach

The approach for clustering window lines (shown in algorithm 9) is based on the default implementation and uses its result as input. The window specific method uses the pre-clustered elements and applies the k-means clustering algorithm [Macqueen1967] based on the lines' orientation to create four line clusters with lines around 0°, 45°, 90° and 135°. This is done due to the assumption that windows that are close to each other have the characteristics that their lines differ in

Algorithm 7: `wallClustering()`: The wall specific clustering approach that uses Distance Map, Zhang-suen thinning and Line Segment Detection to cluster wall lines.

```
Data: walls
Result: all clusters
1  image ← drawlines(walls)
2  distanceMap ← distanceMap(image)
3  thresholdedMap ← threshold(distanceMap)
4  thinned ← thinning(thresholdedMap)
5  lines ← lineSegmentDetection(thinned)
6  clusters ← list()
7  forall line in lines do
8      cluster ← list()
9      boundingbox ← boundingBox(line)
10     forall wall in walls do
11         if containedIn(wall, boundingbox) then
12             add(cluster, wall)
13             remove(walls, wall)
14         end
15     end
16     if notEmpty(cluster) then
17         add(clusters, cluster)
18     end
19 end
20 forall cluster1 in clusters do
21     forall cluster2 in clusters do
22         if contains(cluster1, cluster2) or
             contains(cluster2, cluster1) then
23             cluster1 ← merge(cluster1, cluster2)
24             remove(clusters, cluster2)
25         end
26     end
27 end
```

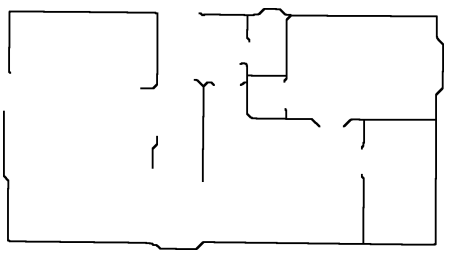


Figure 4: The line skeleton after applying the Zhang-Suen thinning algorithm on the threshold distance map image.

Algorithm 8: `wallPostProcessing()`: Wall specific contour post-processing, that removes small walls and merged overlapping walls into a single bigger wall using the size of the building plan to calculate the threshold for the merging area.

```
Data: walls, buildingPlanSize
Result: postprocessed wall contours
threshold ← getThreshold(buildingPlanSize)
forall wall in walls do
    rotatedRectangle ← rotatedRectangle(wall)
    if area(rotatedRectangle) < threshold then
        remove(walls, wall)
    end
end
forall wall1 in walls do
    forall wall2 in walls do
        if intersectionRatio(wall1, wall2) > 0.4
         then
            merged ← merge(wall1, wall2);
             remove(walls, wall1)
            remove(walls, wall2)
            add(walls, merged)
        end
    end
end
```

Algorithm 9: `windowClustering()`: Window specific clustering with line outlier removal

```
Data: windows
Result: window clusters
forall window in windows do
    k ← 4
    do
        clusters ← kMeans(window.line, k)
        k ← k - 1
    while clustersAreSimilar(clusters)
    remove(windows, window)
    addAll(windows, clusters)
end
forall cluster in windows do
    weightedCenter ←
     averagePointPosition(cluster.lines)
    averageLength ← averageLength(cluster.lines)
    medianDistance ←
     medianDistance(weightedCenter,
     cluster.lines)
    upperQuantil ← medianDistance * 1.5
    forall line in cluster.lines do
        if distance(center(line), weightedCenter) >
         medianDistance then
            remove(cluster.lines, line);
        end
        if length(line) >= averageLength * 2 then
            remove(cluster.lines, line);
        end
    end
end
```

rotation with a certain threshold. Those clusters are checked on similarity using the average line rotation per cluster. If at least two clusters are too similar (difference < 30°), the k-means clustering is repeated with a smaller number of clusters until all clusters are unique. Afterwards, outliers are removed from the clusters. This is done by two different approaches. One approach removes positional outliers and the other removes length outliers. The first outlier group is identified using the average point position of all lines' start- and endpoints of a cluster as weighted center and the median distance from the start- and endpoints to this center. A line is considered an outlier if the distance of its midpoint to the cluster's weighted center is in the upper quantile of all line distances. In addition to that, lines are removed if their length is at least twice as long as the average line length in the cluster. The resulting and filtered clusters are used as input for the default contour conversion.

4.2 Case Study

In order to evaluate the floor plan transformation, a case study is designed. In this case study we use a CSV representation of a real world 2D floor plan, which is shown in figure 6(c), as well as 100 generated, i.e. synthesized, building plans described in section 4.3, that consists of classified lines, which are then transformed into contour elements. The real world floor plan was scanned and provided with a resolution of 1358×988 pixels. Since the input image is classified before it is used in our clustering approach, it is also pre-processed and a threshold is applied to remove levels of gray and noise.

4.3 Random Building Plan Generator

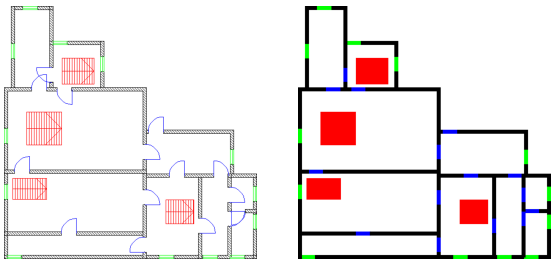
The random building plan generator allows us to generate lines that form a plausible building plan. Within the developed image to 3D building plan pipeline, this approach was initially used to train the classification module to overcome the limitation of having to train and test with the same data source. In addition to that we use the synthesized building plans to test our clustering approach. The architecture of the random building plan generator is split into two parts. At first, we implement a simple random room generator and secondly the room borders are used to place textured windows, walls and doors on them. Additionally, stairs are placed inside these boundaries.

The generation process starts by placing random rectangles inside an image. These rectangles are extended

in width and height as long as they are not touching any other rectangles. Once they hit another rectangle on a specific side, the growing on this side stops and the process is continued with the non-touching sides. The method is applied until the growing of all rectangles has finished. This process can lead to small holes where multiple rooms connect. Nevertheless, this is not an issue as such holes also occur in regular plans for chimneys or air shafts.

After the rooms are created, we start using the lines of the rectangles for further processing. In a first step we determine possible window and wall spots. We evaluate for every line if it is an inside or outside line by checking if it overlaps with a line of any other room. In this case, the overlapping area of one of the lines is removed and both lines are marked as inside lines. The region of the line is then contained in two rectangles and is marked as possible place for a door. On outside lines the entire line is marked as possible position for a window. In the next step the determined positions are used to place randomized doors and windows, as well as one entrance door that is generated at a window position. After that, we have a set of lines for doors and window. All remaining lines are marked as wall lines. For generating the stairs, we randomly add rectangles inside room areas.

In the final step of the building plan generator we apply a texture to all these lines and add a spacing wherever necessary, so that there are no overlapping areas. In the end this results in an automatically generated building plan as shown in figure 5 (a). In this image black lines represent walls, red line are stairs, green lines are windows and blue lines are doors. The final result may contain some unrealistic rooms, that have an odd aspect ratio. Nevertheless, this is sufficient for the purpose of evaluating the clustering process of classified lines. These building plans are then exported as CSV-file containing all the lines and their corresponding type. In addition to the CSV output, we also generate the corresponding contour elements show in figure 5 (b) that are used as a ground truth for the evaluation.

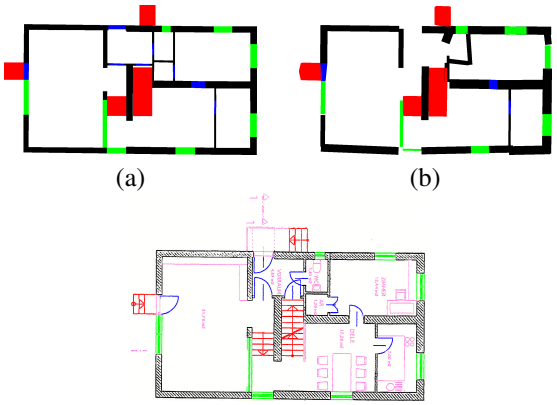


(a) (b)

Figure 5: (a) Sample of an auto generated building plan using Random Building Plan Generator and (b) the corresponding filled contour image, used as ground truth result.

5 EVALUATION

The presented approach is expected to be feasible for clustering single lines. In order to evaluate the correctness of these clusters, they are represented as filled polygons and are compared to our predefined ground truth. Such a comparison is shown in figure 6. Where (a) shows the ground truth that was manually created using an image manipulation program and is compared to (b) the automatically generated result. The evaluation is performed pixel-wise. This means, that we compare the color of each pixel from the automated approach with the ground-truth's counterpart.



(a) (b)

(c)

Figure 6: (a) The manually created ground truth, (b) the transformation result of the real world show in (c). Black segments represent walls, red segments are stairs, blue segments are doors and green segments represent windows.

The described transformation is executed on the classified floor plan shown in figure 6(c) and results in the clustered plan shown in figure 6(b). Together with the manually transformed plan as ground truth (shown in figure 6(a)), the transformation is evaluated. In addition to this real world model, we use the random building plan generator to additionally create 100 equally sized building plans together with their corresponding ground truth contour representation.

Using the ground truth plans and the results of the transformation process the differences are calculated, which is shown for the real world example in figure 7 with figure 6(a) and (b) as input. Table 1 shows the confusion matrix for the various types between the ground truths and our process results for all transformations. It shows, that the transformation works for most types with an accuracy of $> 90\%$. Nevertheless, the transformation fails on doors and cannot transform them. As the table shows, some door lines result in background pixels. The reason for that is, that if the specific transformation step cannot extract the door with a high confidence no door at the desired position is generated, as described in section 4.1.1.

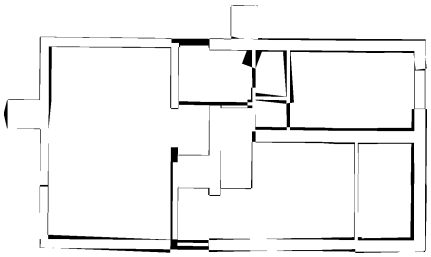


Figure 7: Difference image of the manually transformed plan shown in figure 6 (a) and the automatically transformed plan shown in figure 6 (b). Black pixels representing differences between the compared images and white pixels are equal in the images

		Automatic Transformation				
		Wall	Stair	Window	Door	Background
Ground Truth	Wall	**92.22%**	0.00%	0.63%	0.09%	7.06%
	Stair	0.04%	**98.50%**	0.00%	0.00%	1.46%
	Window	0.25%	0.00%	**99.75%**	0.00%	0.00%
	Door	0.79%	0.00%	0.00%	**7.90%**	91.31%
	Background	1.43%	0.00%	0.10%	0.04%	**98.43%**

Table 1: Confusion matrix for the evaluation of the transformation with normalized values for each type.

6 RESULTS

6.1 Accuracy

The accuracy of the single classes can be seen in the normalized confusion matrix in table 1. Beside the background, which has a very high accuracy due to the total amount of background pixels, the stairs and windows perform even better. Those are followed by walls, which do have a slightly worse accuracy. With only an accuracy of 7.9% doors are the least correctly transformed category. As described in section 5, this happens due to the fact that the transformation does not create a door if the confidence is too low. Overall we can state that the transformation accuracy is with $> 90\%$ for walls, stair and windows sufficient, only for door with around 8% it is not sophisticated for the tested example.

6.2 Performance

In contrast to the accuracy evaluation that is done on equally sized plans, the performance evaluation uses different sized building plans. For this purpose plans with diverse numbers of lines as well as variant widths and heights are generated using the random building plan generator.

Results of the performance measurements are shown in table 2. The experiments are executed with *10* warmup tests followed by additional *10* measured tests, each. The results in the table are averaged values. The results show that the performance scales with the size of the plan in a nearly linear manner. This means, that if the plan size is doubled, the run-time is around twice the time. This is also proven by the strong *Pearson* correlation between area and run-time of around *.996*, as well as in the run-time chart which can be seen in figure 8.

		Plansize (in pixel)			
Rooms	Lines	width	height	area	run-time [ms]
5	686	603	603	363609	725
6	852	603	603	363609	740
7	1011	603	603	363609	1019
8	1248	803	803	644809	1315
9	1375	803	803	644809	1393
10	1741	1029	1003	1032087	2522
12	2428	1303	1303	1697809	3847
14	2581	1529	1503	2298087	4459
16	3600	1703	1703	2900209	7809
18	2960	2003	2003	4012009	10519
20	3776	2503	2503	6265009	15743

Table 2: Performance of the transformation in ms compared to the number of lines, number of rooms and size of the building plan.

It is also affected by the number of lines in the plan. If the number of lines is increased but the size is kept constant, the run-time also increases. This does not have a big impact on the run-time, which can be seen in the slightly lower correlation of around *.896*.

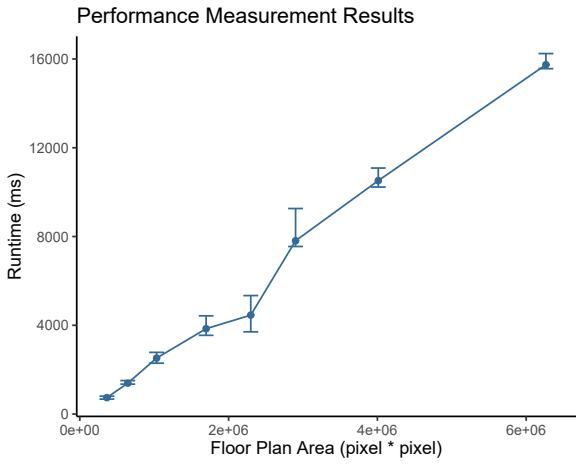


Figure 8: The run-time of the transformation grows linear with the area size of the plan.

7 CONCLUSION

In section 4.1 we answered our research question: How can classified lines be transformed to contour elements using clustering approaches?, and presented an approach on how to automatically cluster lines in the context of a 2D building plan. This concept uses algorithms from the field of computer vision and machine learning.

We evaluated that concept in section 5 using a case study consisting of a single real world floor plan and 100 synthetic building plans.

For this case study, the accuracy was evaluated in section 6.1, which showed an accuracy of $> 90\%$ for window, walls and stairs and around 8% for doors. In addition to that, a performance test was done in section 6.2. The test showed that the average run-time scales linearly with the area of the building plan.

In the future we want to extend the evaluation with a higher amount of real data as well as a higher variety of building plans. Moreover, we aim to improve the transformation accuracy for door elements. This can e.g. be achieved by changing the classification to have separated classes for normal and overlapping doors, which would then give the possibility to address these two tasks differently in the transformation. In addition to that we want to do an evaluation on scaled building plans. As we have seen a near linear scale in run-time compared to the size of the plan this should lead to a increase in performance.

8 REFERENCES

[AhSoon2001] Ah-Soon C. and Tombre K. Architectural symbol recognition using a network of constraints, in Pattern Recognition Letters Vol. 22, pp. 231-248, 2001.

[Arthur2006k] Arthur D. and Vassilvitskii S. k-means++: The advantages of careful seeding, 2006.

[Bresenham1965] Bresenham J. Algorithm for computer control of a digital plotter, in IBM Systems Journal Vol 4, pp. 25-30, 1965.

[Canny1986] Canny J. A Computational Approach to Edge Detection, in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 6, pp. 679-698, 1986.

[Dosch2000] Dosch P, Tombre K., Ah-Soon C. and Masini G. A complete system for the analysis of architectural drawings, in Int. Journal on Document Analysis and Recognition Vol. 3, pp. 102-116, 2000.

[Duda1972] Duda R.O. and Hart P.E. se of the Hough Transformation to Detect Lines and Curves in Pictures, in Comm. ACM. 15: 11â€“15, 1972.

[Eberly2015] Eberly D. Minimum-area rectangle containing a set of points, 2015.

[Gerstweiler2018] Gerstweiler G., Furlan L., Timofeev M. and Kaufmann H. Extraction of Structural and Semantic Data from 2D Floor Plans for Interactive and Immersive VR Real Estate Exploration, in MDPI AG Vol. 6, 2018.

[Lee2006] Yun-Seok L., Han-Suh K. and Chang-Sung J. A straight line detection using principal component analysis, in Pattern Recognition Letters vol 27(14), 2006.

[Lewis1998] Lewis R. and SÃ©quin C. Generation of 3D building models from 2D architectural plans, in Computer-Aided Design Vol. 30, pp. 765 - 779, 1998.

[Liu2017] Chen L. and Jiajun W. and Pushmeet K. and Yasutaka F.. Raster-to-Vector: Revisiting Floorplan Transformation, in Conf. proc. of IEEE International Conference on Computer Vision (ICCV) 2017.

[Lu2007] Lu T, Yang H. Yang R. Cai S. Automatic analysis and integration of architectural drawings, in Proc. of Int. Journal of Document Analysis and Recognition (IJDAR) Vol. 9, pp. 31 - 47, 2007.

[Macqueen1967] MaqQueen J. Some methods for classification and analysis of multivariate observations, in Proc. of the fifth Berkeley symposium on mathematical statistics and probability Vol. 1., pp. 281-297, 1967.

[Or2008] Or S, Kin-Hong Y. and Ming M. Abstract Highly Automatic Approach to Architectural Floorplan Image Understanding & Model Generation, 2008.

[Pointner2018] Pointner A., Krauss O., Freilinger G., Strieder D. and Zwettler G. A. Model-based image processing approaches for automated person identification and authentication in online banking, in Proc. of the EMSS2018, 2018.

[Preparata1985] Preparata F. and Shamos M. Computational geometry: an introduction, 1985.

[So1998] So C. Baciu G. and Sun H. Reconstruction of 3D virtual buildings from 2D architectural floor plans, in Proc. of the ACM symposium on Virtual reality software and technology, 1998.

[Stojanovic2019] Stojanovic V., Trapp M., Richter R. and Döllner J. Generation of Approximate 2D and 3D Floor Plans from 3D Point Clouds, in Proceedings of VISIGRAPP, 2019.

[VonGioi2012] Von Gioi R., Jakubowic J., Morel J. and Gregory R. LSD: a Line Segment Detector, in Image Processing On Line Vol. 2, pp. 35-55, 2012.

[Yin2009] Yin X., Wonka P. and Razdan A. Generating 3D Building Models from Architectural Drawings: A Survey, in IEEE Computer Graphics and Applications Vol. 29, pp. 20 - 30, 2010.

[Zeng2019] Zeng Z., Li X., Yu Y. and Fu C. Deep Floor Plan Recognition Using a Multi-Task Network With Room-Boundary-Guided Attention, in Proc. of. Int. Conference on Computer Vision (ICCV), 2019.

[Zhang1984] Zhang T. and Suen C. A fast parallel algorithm for thinning digital patterns, in Communications of the ACM Vol. 27, pp. 236-239, 1984.

[Zwettler2010] Zwettler G.A., Backfrieder W., Swoboda R. and Pfeifer F. Fast Medial Axis Extraction on Tubular Large 3D Data by Randomized Erosion, in Springer Lecture Notes CCIS, Springer Vieweg, pp. 97-108, 2010.